2. How does command injection work?

How Command Injection Works

Command injection occurs when an attacker manipulates an application's input to trick it into running unintended system commands. This happens in applications that run user inputs directly in command-line statements, especially when user input isn't sanitized or validated. Here’s how it typically plays out:

1. **Vulnerable Input Handling**:
   - When an application uses a command-line statement that includes user input, it may concatenate or directly insert that input into a shell command.
   - For example, if an app takes a filename as input and runs a system command like `cat <filename>`, it might be written in code like this:

     ```
     os.system("cat " + user_input)
     ```
2. **Exploitation with Special Characters**:
   - Attackers can take advantage of **special shell characters** (`;`, `&&`, `||`, `|`) to add commands after the legitimate input.
   - These characters allow attackers to terminate the intended command and execute a new one.
3. **Example Attack**:
   - Suppose an application asks for a username and directly inserts it into a shell command:

     ```
     os.system("echo Hello, " + username)
     ```
   - If a user enters `john`, the command would run as:

     ```
     echo Hello, john
     ```
   - However, if an attacker enters `john; rm -rf /`, the command becomes:

     ```
     echo Hello, john; rm -rf /
     ```
   - Here, the `rm -rf /` command would execute after the first `echo` command, which could delete system files (in a real attack, they’d target specific files to avoid obvious destruction).
4. **Types of Command Injection Techniques**:
   - **Chaining commands** with `;` or `&&` to run additional commands.
   - **Piping** commands with `|` to redirect output into malicious commands.
   - **Subshell execution** using backticks (`` ` ``) or `$()` to execute commands within commands.
5. **Blind Command Injection**:

- In cases where the app doesn’t show output (e.g., web servers), attackers use **blind injection** by testing for time delays or observing indirect effects (like server load or resource status) to see if a command ran.

Real-World Example

Say a web application lets users "ping" a server by entering a domain name. The app might run this command:

```
ping -c 4 example.com
```

If the input isn’t validated, an attacker could enter:

```
example.com; ls /etc/passwd
```

This would execute `ping` on `example.com`, followed by `ls /etc/passwd` to list sensitive files.

Preventing Command Injection

1. **Avoid Shell Commands**:
   - Use **API functions** or libraries that interact with the OS securely, instead of directly calling shell commands.
2. **Input Validation and Sanitization**:
   - Limit inputs to expected characters (like alphanumeric characters) and strip or encode any special characters.
3. **Parameterized Statements**:
   - Some programming languages support parameterized statements, which keep user inputs separate from the command structure, reducing injection risks.

Command injection is powerful and dangerous because it can give attackers direct control over a system.